PORTABLE DISTRIBUTED SYSTEM SOFTWARE

FOR

MICROCOMPUTERS

A Thesis

Presented to

The Chinese University of Hong Kong

In Partial Fulfillment of the Requirements

For The Degree of Master of Philosophy

by

KWAN Kar Kin, Karl

May 1987

ABSTRACT

This thesis describes research in the area of portable distributed system software and, especially for microcomputers.

The research objective is to examine the issues of portable system software in a distributed environment. The goal is to modify an existing portable system , Edison, to become a portable distributed system.

The Edison system has been ported to two different types of microcomputers. A simple ring network has been designed and implemented to interconnect the above machines to form a loosely-coupled distributed hybrid system.

Based on the porting experience, it is found that the overall portability of the original system can be improved in various ways. One of the major problems which was encountered during the porting procedure was the transfer of existing valuable software and data to a new machine. However, when the portable system is modified to become a distributed system, the data transfer problem disappears. Therefore, the overall portability of a distributed system is relatively improved. In addition, in order to achieve a higher degree of overall portability, a simplified Unix-like

file system has been designed for the hybrid system. It is concluded that the overall portability of the hybrid system is far greater than the original one.

The major advantage of the portable distributed system is that executable codes can be directly shared by heterogeneous microcomputers. Thus, the software development cost of this system would be extremely economical.

Looking at other kinds of system software for microcomputers, it is found that the portable distributed Edison system can be applied to many different microcomputers. In general, it is believed that portable distributed system can be viable and cost-effective for modern microcomputers.

ACKNOWLEDGEMENTS

I am extremely grateful to Dr. Kam Wing Ng and Mr. Henry Lam for their encouragement, patience and expert guidance during the course of this work. Special thanks should also go to Dr. Ng for his constructive reading of this dissertation. I wish to thank the staff in the Microprocessor Laboratory of the Chinese University of Hong Kong, too, for their helpfulness in many ways.

Finally, I would like to express my sincere thanks to Intel Corporation (HK) Limited for providing many helpful comments and suggestions during the working period. The support of the Intel Corporation(U.S.A.) in this project (a system 286/310 and related hardware and software) is also gratefully acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION

In general, system programmers may have no alternatives other than using assembly level programming languages to implement system programs in order to squeeze out the last bits of efficiency from the underlying machines. Unfortunately, the shortcomings of using assembly languages for system software implementations are too well-known : long development time and lacking of flexibilities. Furthermore, the maintenance cost for this kind of system software are usually enormous. The problem of maintenance was not so critical in the past in the area of microcomputers since the system programs of such machines were usually quite compact, thus requiring much less effort to maintain. However, microcomputers are becoming more advanced and complex nowadays and the maintenance of system software for microcomputers becomes no longer an easy task. Therefore, the portability of system software is becoming an important issue in system software design. Another important issue in system software design is the reliability of system software, and there is a close relationship between these two issues.

1.1 RELIABILITY AND PORTABILITY

Software reliability is extremely important. Software-failure -- that is, the unpredicted behaviour of some part of the system - may have catastrophic effects. A small system bug of a microcomputer, for example, can cause the loss of life and property when it is used to control a chemical process in a petroleum refinery[Caro80]. Unfortunately, the reliability of a newly released microcomputer system software may be overestimated. For instance, the Xenix system released for Intel 286/310 SBC(Single board computer) had been updated at least twice since software bugs were found after the customers received the first version of the Xenix system. Based on experience, the sources of computer system failure include hardware/software design errors,software coding errors, and human errors[ALAN87]. Software coding errors are easier located and design errors can be easier avoided if the system software is written in a high level language[Davi80]. Actually, this is one of the reasons why we need high level languages. Therefore, a way to improve system reliability is to use a high level language to implement the target system software. For example , the MUSIC and SOLO operating systems are written in Concurrent Pascal[Norw79]. Such type of systems is believed and proven to be easier to

maintain and more reliable. Furthermore, the portability of such systems is generally higher , and they can be easily installed on machines other than the original one. These systems are so called portable operating systems. It is obvious that portable operating systems can have a better chance of providing software reliability since these systems have been installed and tested on different machines. In addition, an advantage of the use of portable software is 'people portability'[Pete83]. The implication is that computer personnel would become more reliable and fewer human errors would occur. In short, portable systems usually provide a higher degree of reliability.

1.2 DISTRIBUTED PROCESSING AND PORTABILITY

There are various interpretations of distributed processing. A definition which has a business focus may be like : 'Distributed processing is putting computer power where the user and problems are'[Hami79]. Another interpretation of distributed processing may refer to a working environment which is constructed by having more than one relatively small and independent computing machines. In this environment, independent machines will cooperate closely to provide a single logical service[John81]. The major advantage of this working environment is providing load sharing and resource

sharing. In this thesis, we will use the latter interpretation in our discussions.

1.2.1 Distributed Operating System

In recent years, researches on distributed system are being undertaken by many research groups. These projects are in various stages of development. Some of them have actually been implemented while others are just paper designs[Andr85]. As mentioned above, the goal of these distributed systems is to provide a resource sharing working environment in general. The domain of resources may include the processing power. For example, one of the machines in the distributed system may activate a remote procedure call(RPC) which will be executed in a remote machine. That is, a machine(client) sends a request and blocks until the corresponding machine(server) sends back a reply. The idea of RPC is to make the semantics of inter-machine communication as close as possible to normal procedure calls because the latter is well understood ,and it is useful as a tool for dealing with abstraction[Andr85]. Since these distributed systems are constructed in a decentralized manner , a conventional operating system software is usually not able to serve the needs of such working environments. Therefore, we need a new kind of operating system for the decentralized computing systems. A

general name of this kind of operating system is 'distributed operating system'. A distributed operating system is one that looks to its a users like a conventional centralized operating system but runs on multiple, independent computational elements[Andr85]. For instance, the Cambridge Ring[Need82] , Medusa[John81] , and Ameoba[Andr85] are all experimental distributed operating systems.

Although there are different approaches to the design and implementation distributed operating systems, a basic rule followed by these systems is to ensure user transparency. This means that the distribution of computational elements must be hidden, otherwise the system cannot claim that the corresponding collection of computing machines is providing a single logical service.

1.2.2 Network Computing Systems

Nowadays , the trend of distributed systems is towards interconnected sets of dissimilar hardware systems. Heterogeneity is often unavoidable, and the advantage of such systems is that new types of computing machines may be added to the distributed working environment as required. Such flexibility is very attractive, especially in the domain of distributed

microcomputer systems due to the fact that the microprocessor technology is advancing so quickly. In fact, new microcomputers are introduced to the market every few months. The main problem in a heterogeneous distributed system is the incompatibilities between different hardware systems. Code sharing and remote procedure call become almost impossible to be carried out in heterogeneous systems since different machines may have different internal data representations and individual instruction sets. One of the solutions towards this problem is by using the client and server approach[Davi87] -- stubs are created for each client and server, the following figure illustrates how RPC usually works in this kind of systems.

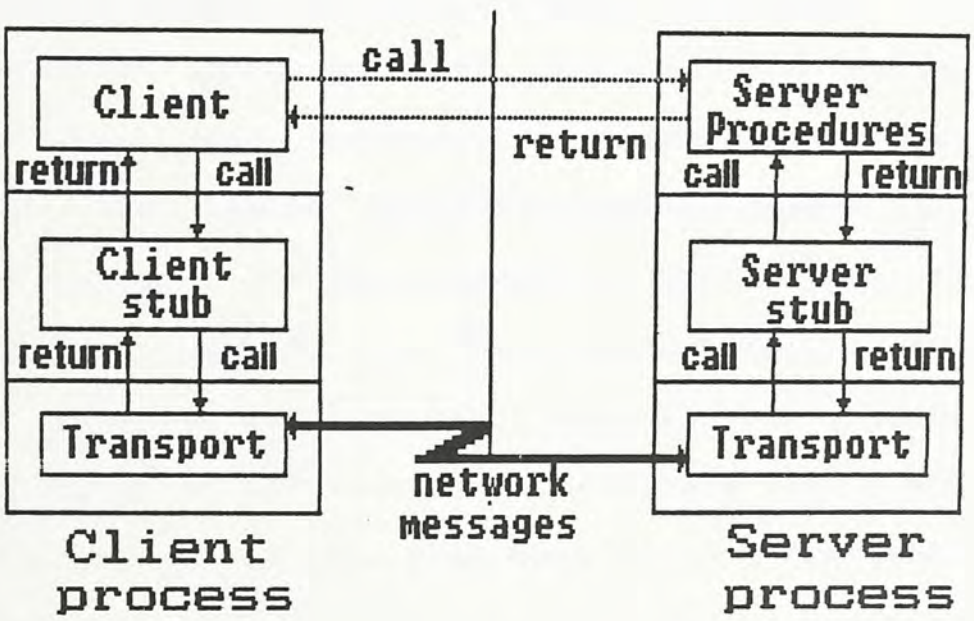


Figure 1.1 Remote Procedure Call

The main duty of the stubs is to perform the translations. At the caller machine , the client stub collects the parameters of the procedure call and packs them into a message in a standard format. It then sends the message to the remote machine and blocks, waiting for the reply. At the remote machine, the server stub receives and unpacks the incoming message to get the parameters, the server stub then makes a local call with the parameters. The remote procedure is thus called locally. The result of the procedure call follows an analogous path in the reverse direction[Andr85]. Interface description languages(IDLs) are often a basis for generating stubs[Davi87]. Usually the IDL contains constructs for describing the data types, functions, and procedures associated with a remote interface. It is a declarative language and contains no executable constructs. For instance, a commercially available system, the Apollo Network Computing System is using this approach. In this approach, RPC service can be carried out easily, but the distribution of executable codes is still not possible. That is, an object code file which can be executed in machine A is not able to be executed in machine B if they are not the same type of machines although they are both interconnected within a so called distributed computing system.

One of the characteristics of portable system software is that it can be executed on different hardware systems. It seems obvious that if the ideas of portable system and distributed system are combined to form a portable distributed system, the sharing of executable object codes(programs) can be easily achieved. If the sharing of programs is allowed in a heterogeneous system, software developement will become more economical.

Because of the above reasons, the research on portable distributed system software was started. The aim of the research is to construct a particular heterogeneous microcomputer system which can support the code sharing service.

1.3 THE APPROACH

Since many reliable portable systems have already been developed, our approach was first choosing one of the existing portable systems as our hybrid system. Hands-on experience of porting issuses was expected to be obtained by actually porting the hybrid system to two different microcomputers.The portable system which was chosen is the Edison System designed by Per Brinch Hansen in 1982. The target machines are Intel 86/310 and 286/310 systems. The next step was to modify the

original system to work on a distributed environment. The reasons why the Edison system was chosen and the details of the modification will be discussed in the next few chapters.

1.4 THE OBJECTIVE

The main objective of this research is to identify issuses of portable system software in a distributed environment. The feasibility of such kind of system will be examined based on an actual implementation. Furthermore, the problems that a portable distributed system implementator may be faced will be identified. We do not expect our hybrid system to be the best portable distributed operating system, rather , it will serve as a starting point. Based on the findings of this research , better portable distributed systems are expected to be designed. Apart from this, the hybrid system can serve practical uses. The hybrid system was originally designed for teaching purposes. It can provide a low-cost programming environment for computing courses such as compiler construction, system programming and computing networks.

1.5 OVERVIEW

The remainder of this thesis is presented in five chapters. The next chapter describes the original Edison System. In the third chapter, the actual procedures of system porting is presented. The basic Edison Ring structure is discussed in the fourth chapter. The fifth chapter presents the basic design of the new distributed Edison system, and the last chapter contains the conclusions of this thesis.

CHAPTER 2.

THE EDISON SYSTEM

Beginning from the mid 70's, many experimental portable languages/systems have been designed and implemented[Pete84]. The interest in portable system software was originally motivated by problems arising in maintaining several different minicomputer systems[Davi82]. The benefit of implementing portable system software is the ease with which such software can be used in different machines, the 'ease' at least relative to implementing from scratch a new software system for each machine. Thus, the costs of implementing and maintaining separate software systems for each machine are reduced significantly. Furthermore, the portability of application programs running under a portable operating system is itself improved since the system interfaces are the same on all target machines[Pete84]. In general , the benefit of 'people portability'[Pete84] will also be achieved in portable operating system, the difficulty of interfacing with the system software is avoided and much of the training of computer personnel which are specific to particular tailor-made system is also preserved.

One important issue that must be considered is the trade-off between system flexibility and machine

efficiency. It is a common idea to implement portable system software based on the interpretation of intermediate languages. Inevitably the execution speed of intermediate code is slower than tailor-made assembly code. For this reason, it seems that portable operating system can never fit into the old-time low speed 1 MHz microcomputers, thus, microcomputer operating systems were largely written in assembly code. High level language based portable systems were mainly designed for minicomputers. In fact, it is interesting to note that during the mid 70's, it would appear that the only high level language that was small enough to fit into a microcomputer was BASIC, one of the earlier portable languages. Consequently, one more advantage of using interpreters apart from providing portability is perceived, that is, interpreters can provide adeqate computing service while occupying only a relatively small size of main memory. The situation has changed in recent years, due to the dramatic competition in the semiconductor industry and advances in silicon technology. Personal computers based on microprocessors become not only more sophisticated but also less expensive. For instance, the iAPX88, 86, 286, 386 microprocessor series of Intel Corporation and the MC68000 microprocessor series of Motorola Corporation do increase the computing power of modern microcomputers to

higher extends. Nowadays powerful microcomputers can support a wide range of applications. Portable software systems which were designed for minicomputers are now available for microcomputers.

2.1 BASIC APPROACHES OF PORTABLE SYSTEMS

Apart from the above mentioned way of implementing portable systems, there are two other approaches[Paul83].

One approach is to use an emulator for an existing computer on a new machine. The operating system and application programs of the old machine can then directly run on top of the emulator. For example, the IBM 7090 and 1401 emulators running on IBM 360 computers[Paul83]. This approach was successful but had a continuous performance penalty.

Another approach is to implement the entire operating system and the compilers in a high level language and to isolate all machine-dependent parts of the operating system and compilers. Then, the code generator for the high level language is modified to generate the machine instructions for the new machine. The machine-dependent parts of the operating system also need to be rewritten. Finally, all of the operating system must be recompiled using the modified compiler to

come up with an operating system that will execute on the new machine. This approach offers the best potential for achieving portable operating systems with the best possible performance. One of the first earlier portable systems, Thoth , an operating system developed at University of the Waterloo used this approach. Most of the Thoth system is written in the high-level language Zed[Davi82]. The major task in transporting the system is to retarget the Zed compiler for the new host computer[Davi82]. The retargeting is done by modifying the fifth pass, the code generator, of the Zed compiler, thus enabling the modified Zed compiler to generate executable code for the new host computer. Therefore, the Thoth system can be ported to the new machine by half bootstrap once the retargeting is complete[Pete84]. A small number of machine-dependent routines that must be written in assebmbly code for each machine can be easily produced by using an 'assembler-generating kit', therefore, the development of this part of the coding does not depend on the availability of a suitable assembler for the target machine[Pete84].

Comparing the above three approaches, it is obvious that the benefit of adopting the emulator approach is to obviate the need to recompile all the program sources of the old machine. In addition, the

entire emulator itself may be implemented in a high level language, thus ensuring the reliability. On the other hand, the Thoth approach may offer the best possible performance for the new system, but it is not able to provide direct code sharing capability as different Zed compilers generate individual machine-dependent instructions.

The use of an interpreter seems to provide a compromise. In this approach, an instruction set for a virtual machine is created. The instruction set is normally "higher level" then assembly code and easier to program. An operating system written in this instruction set can be executed by a machine-dependent interpreter. Porting can be done by simply rewriting the interpreter, and all the system software and application programs of the old machine can be run directly on the new environment. The performance penalty of interpretation still has to be paid, but the fact that the interpreted "instructions" are higher level may have reduced this effect[Paul83].

2.2 HYBRID SYSTEM

There are several reasons in choosing Edison as the hybrid system of this research project. The main reason being that the portable Edison system has

potential to allow direct code sharing in a distributed environment since the system design is based on the interpreter approach.

2.2.1 The Portability Goal

The main goal of the original Edison system is to provide a better working environment which is efficient, reliable, extensible and portable for the computer user. The original Edison system consists of a machine-dependent kernel, an operating system written in the Edison language and a four pass portable Edison compiler(it is also written in the Edison language). The small kernel mainly serves as a threaded code interpreter for the Edison intermediate language (which is generated by the Edison compiler) and only the kernel of the system is not portable. The following figure shows the basic idea of the Edison threaded code interpreter.

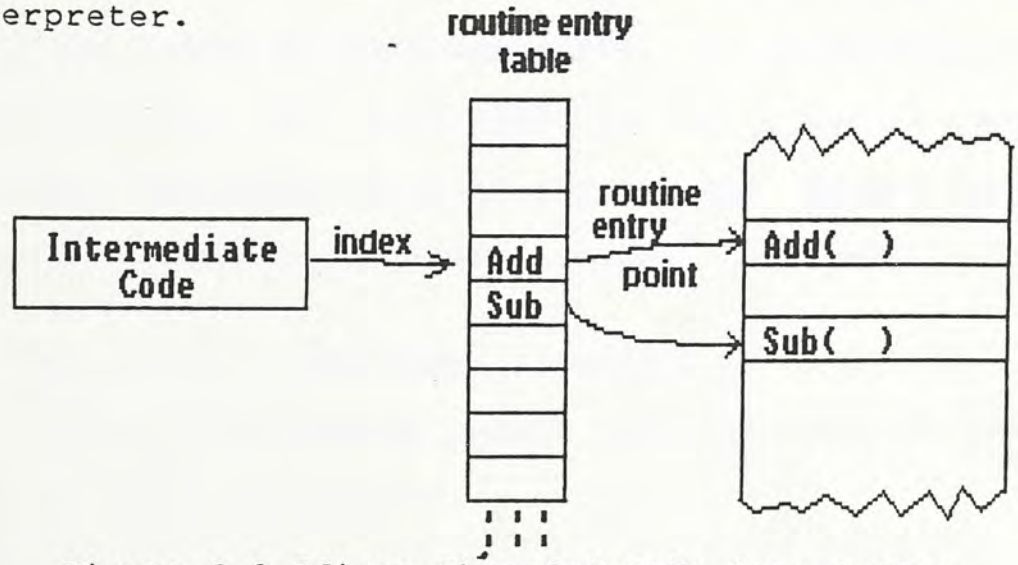


Figure 2.1 Edison Threaded Code Interpreter

An intermediate code instruction is interpreted by a machine code piece. A routine entry table is used to store the addresses of all the code pieces. Each of the intermediate code instructions is actually an integer value that represents an index of the routine entry table, from which the associated machine code piece can be located, thus the interpretation can be carried out. This indirect addressing method provides an efficient way of interpretation[Brin84].

2.2.2 The Edison Language

The Edison system itself is written in a high level language which is also named Edison. This Edison language was developed based on Sequential Pascal and Concurrent Pascal by Per Brinch Hansen[Brin83]. Since the Edison language was designed for the purpose of portable system software programming, it has adequate features common to both Sequential and Concurrent Pascal which enable the programmer to use it for a **modular approach** to programming. Besides, the Edison language also provides the following special features :

a) **Cobegin, Also** and **When** statements

These statements permit the programmer to define and execute parallel processes.

b) Type Conversion

Flexibility is provided to allow operations to be applied to variables of different types as long as they occupy the same size of memory. Complex data types such as the variant record structure can be implemented in Edison by using this feature.

c) Constructors

Constructors allow easier data initialization and assignment for complicated data types.

In short , Edison is a simple to learn , easy to use high level system programming language.

2.2.3 Edison Operating System

The Edison operating system program consists of only about 1200 lines of Edison program text. The system provides an interactive single user working environment for the user. A complete disk file management system and command line interpreter are also built in. There are also about thirty system calls defined in the operating system program text. These system calls provide basic input/output operations and they can be called by upper level application programs. Since the operating system is implemented in the Edison language, the system itself can be easily modified. This property provides a higher

degree of flexibility for implementing hybrid systems.

2.2.4 System Configuration

The original Edison system was first implemented on a PDP 11 microcomputer with the following capabilities[Brin83]:

28K word store
Dual 8" floppy disk drive(RX01 or RX02)
ASCII terminal (VT52 or VT100)
ASCII printer(Diablo 1610 or Xerox 1740)

The associated floppy disk has 77 tracks, each of which is divided into 26 sectors of 64 words each. The floppy disk is treated as an array of sectors with sector number in the range of 0 to 2001[Brin83]. The original operating system was written in a way to cooperate with this system configuration. If the system is going to be transported to a new machine, certain parts of the operating system may need to be modified to suit the new configuration.

2.2.5 Edison Kernel

As mentioned above, the only machine dependent part of the Edison system is the kernel program, and this kernel is usually implemented in assembly language.

Basically the kernel program can be functionally separated into three parts :

i) Bootstrap loader

ii) I/O device drivers

iii) Threaded code interpreter

In the original Edison system. The executable kernel code(machine dependent) is stored in sector 26 to sector 89 of the system disk. When the system is booting up, the first part of the kernel code will be loaded into ram first and then this part of the kernel takes control and it will load in the rest of the kernel code. After this is done the kernel will then load in the Edison OS code(Edison intermediate code) which is residing in disk sector 90 to 195. After the system is brought in, the kernel will then serve as the threaded code interpreter, that is, it will fetch the next avaliable intermediate code from memory and treat the code as an indirect branch index to jump to the corresponding kernel routine to carry out the operation. The kernel also provides several standard kernel calls to allow user programs to access the terminal, printer, and the floppy disk based file system.

2.2.6 Memory Management

The structure of memory management in the Edison system is simple. The addressing space is only 64K bytes, which allows the Edison system to be ported to almost any microcomputers without worrying that it is too large to fit in the target machine. The kernel program is located at the lower end of the memory. Initially, an Edison program is executed as a sequential process with a single variable stack and a single program stack.

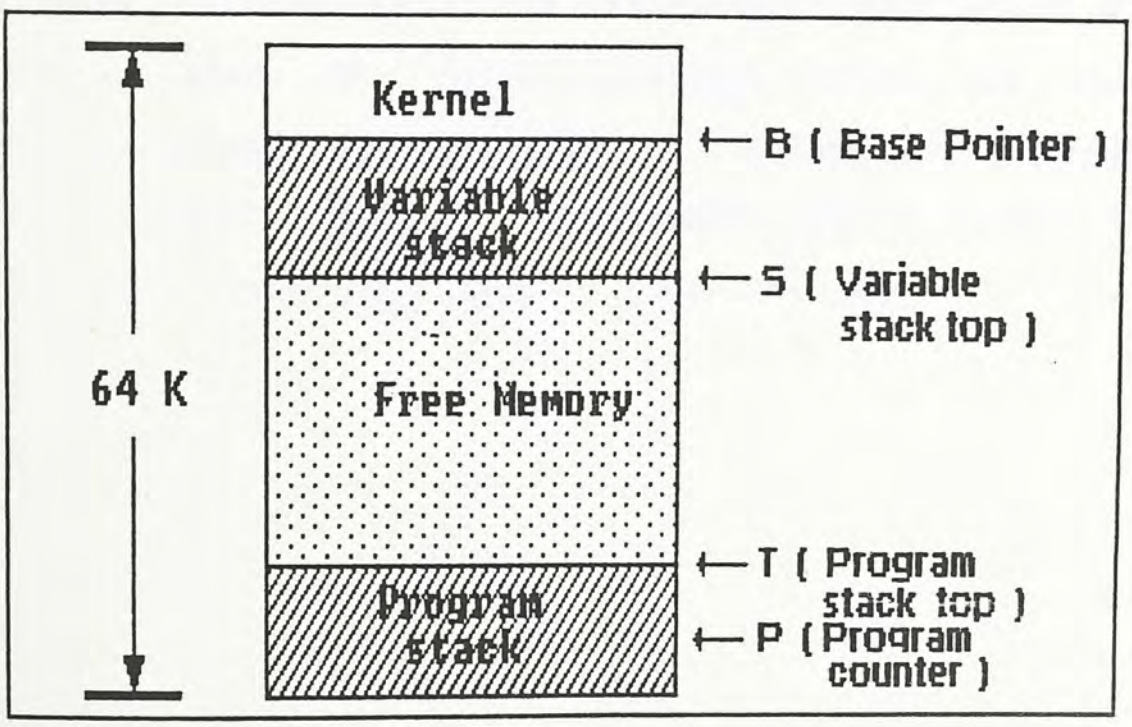


Figure 2.2 Edison Store Configuration

Figure 2.2 shows the store of a single-processor system which holds a kernel and the two stacks. The stacks grow towards one another in the free space between them. After the system is booted up , the operating system itself will be the first program which is loaded into the program stack region. The variables of the operating system are stored in the variable stack. Application programs which are activated by the operating system will be treated as Library procedures, and they will be loaded into the free space area of the memory store. These Library procedures will follow the compile_and_go fashion. After the execution of the program is completed, its associated memory space will be released. The following figure shows the store configuration when an application program is just loaded into memory. Actually the kernel program treats the operating system as a library procedure too.

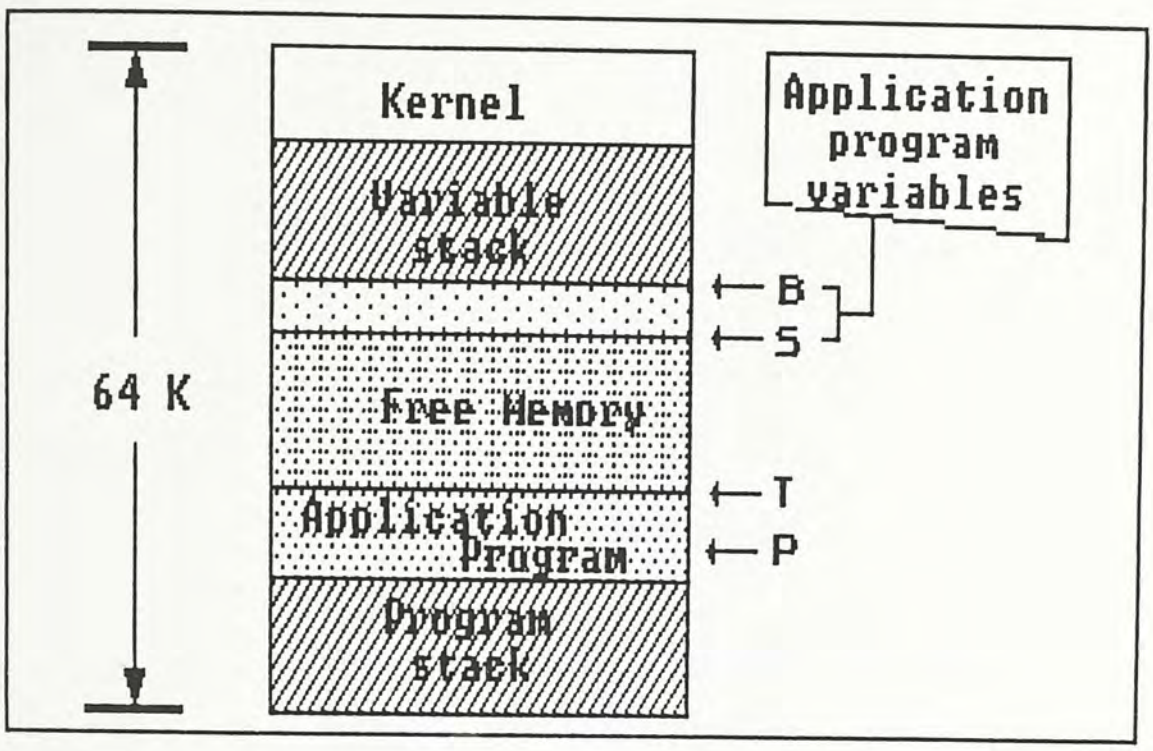


Figure 2.3 Edison System With Application Program

2.3 SUMMARY

The beauty of the Edison system lies in its simplicity and is very suitable for a few man-months project due to its compact size. In addition, the library procedure feature can be well fitted to a code sharing distributed environment. For example, a single copy of the text editor code file may be called up and loaded to heterogeneous host machines as a library procedure.

Experience of porting the Edison system to different microcomputers will be presented in the next chapter.

CHAPTER 3.

TRANSPORTING THE EDISON SYSTEM TO MICROCOMPUTERS

In this chapter, the porting procedures of the Edison system will be discussed in detail. Based on the porting exercises , issues of system portability will be investigated.

3.1 TARGET MACHINES,

The original Edison system was first developed on a PDP 11/23 microcomputer equipped with a terminal , a printer and two 8" floppy disk drives. In this research project, two different types of microcomputers were chosen for the target machines. The first one was an Intel 86/310 based on a 5MHz iAPX86 microprocessor. It has 640K bytes of memory on board, a serial terminal port , a parallel printer port and is also equipped with one 720K bytes and one 360K bytes 5.25" floppy disk drives. This target machine was chosen mainly for the popularity of the iAPX88/86 microprocessor series.Since the hardware configuration of the 86/310 was similar to the original Edison system, it was not necessary to apply much modification to the operating system program.

The following diagram portrays the configuration of the 86/310 single board computer:

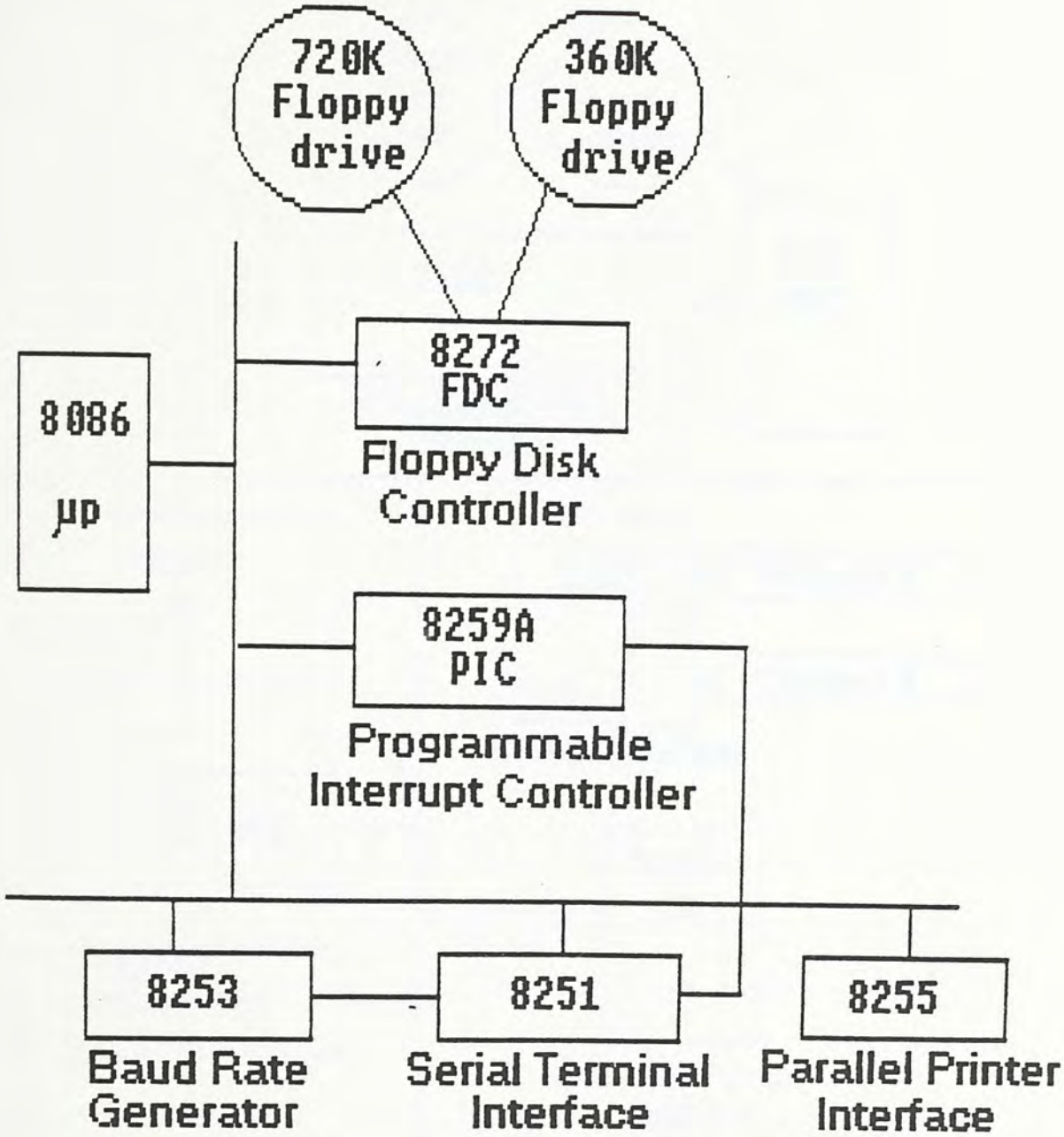


Figure 3.1 Block Diagram Of The 86/310

The other target machine is an Intel 286/310 based on a 6 MHz iAPX286 microprocessor. This machine was chosen because the iAPX286 was one of the most powerful microprocessor in 1985, the time that this project was

started. The following diagram shows the configuration of the 286/310 system.

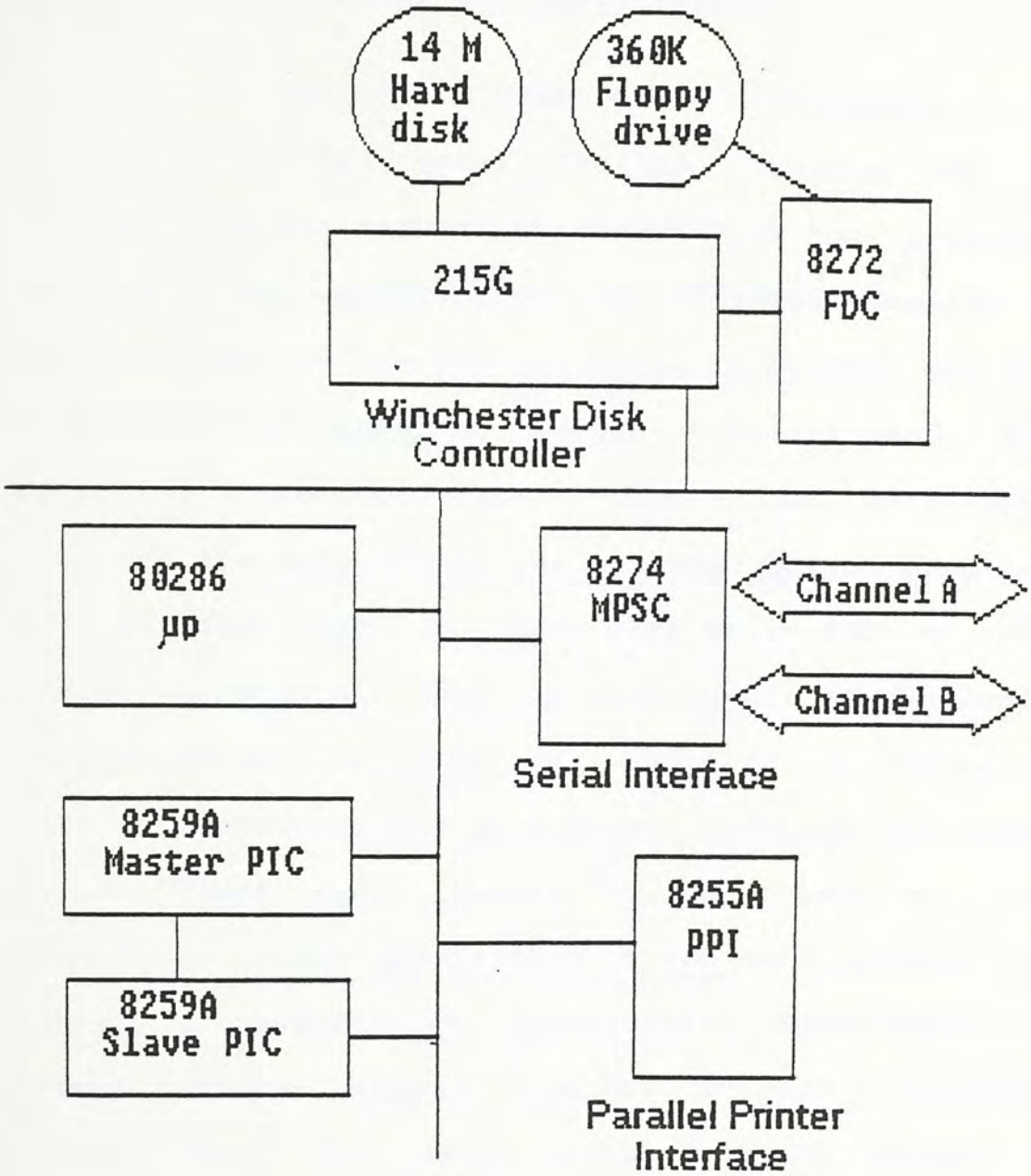


Figure 3.2 Block Diagram Of The 286/310

3.2 PORTING PROCEDURES

The Edison system was first ported to the 86/310 machine, and then ported to the 286/310. The details of the porting procedures are presented in the following

sections.

3.2.1 Edison Compiler For Bootstrapping

Since the system used in the development of this project was the Intel 286/310 (running the Xenix system), it was natural to choose C as the programming language for implementing the Edison compiler. The approach was to use this compiler to compile the source text of the original Edison compiler and Edison operating system programs to obtain the corresponding intermediate code files. The original Edison compiler is a four pass compiler. Each pass is in fact an Edison Library procedure. When the main module of the compiler is activated, it will call and load in these four Library procedures one at a time. Although the compiler allows these four passes to call certain common procedures which are defined in the main module, these Library procedures are functionally independent, the communications between them are through intermediate files. Since the Xenix system does not support any feature that is identical to the Edison library procedure, four independent C programs were written to simulate the functions of these four Library procedures.

The following batch file 'comp' can be used to carry out the compilation process :

```
[  pass1.out $1
   pass2.out
   pass3.out
   pass4.out
   mv  temp2 $2 ]
```

Figure 3.3 Batch File For Compilation Process

The invocation line for the above batch file is -

comp sourcefile destinationfile

The program pass1.out will take **sourcefile** as input and generate an intermediate output file named temp1. Pass2.out will take temp1 as input and generate an output named temp2 and so on. After the execution of the batch file is completed, temp1 will be destroyed and the final output will be stored in **destinationfile.** The shortcoming of this design is obvious since there is no main module to store the common procedures. Therefore , duplications of these procedures appear in all four passes and they make the program texts longer than the original ones. Another problem encountered was caused by the incompatibility between C and Edison. There is no set oriented instructions available in C while the Edison compiler parser uses such kind of instructions excessively. The solution towards this problem was to define a utility module to carry out the set oriented

operations. The data structure for representing enumeration type variable is a 16-byte long array. Each bit of the array represent one element,thus the maximum number of elements in an enumeration type variable is 128, and this corresponds to the definition of enumeration type variable in the Edison language.

The following example shows how a Union operation is done in the Edison bootstrap compiler :

```
union(enum1,enum2)
int enum1[], enum2[];
{  short i;
     for (i = Ø, i < SETLIMIT , i++)   /* SETLIMIT = 8 */
     enum1[i] = enum1[i] | enum2[i];
}  /** end of union **/
```

Figure 3.4 Utility Routine For Union Operation In C

This extra utility module again makes the size of the Edison compiler longer than the original one.

3.2.2 Kernel Simulator On Xenix

After the coding phase of the bootstrap compiler was completed, dozens of small Edison programs were written and fed into the compiler to test the correctness of the implementation. Generally, this debugging method was only sufficient for the testing of pass1, pass2 and pass3. The correctness of the intermediate code generation pass was still questionable. As mentioned by Paul J. J.[Paul83], "a lot of time can be wasted tracing even minor compiler bugs with a not-yet-working operating system". At this point, an additional phase was designed to add to the porting procedures in order to make the debugging easier. The extra work was an implementation of a kernel program which could be run on top of the Xenix system. This kernel program was also written in C and the bootstrap loader part of the kernel could be omitted. Therefore, this kernel contained only the threaded code interpreter and the I/O drivers. One of the advantages of using C to implement the Edison Kernel is that C could construct jump table easily to carry out the threaded code interpreting operation. The C code in the next page shows how to construct the jump table.

```
long  (*jmptable[MAXOPCODE])  () ;  /* Declare array  of
                                       pointers */

/* MAXOPCODE equals to the total number of threaded code
   routines */

  /*************************************/

init_table() /* routine to initialize the jump table */
{
      jmptable[Ø] = add;     /* put routine address  into
                                the table */

      jmptable[1] = alsox; /* add and alsox are routines
                              names */
      :              :          :
      :              :          :
}

  /*************************************/

main()

{  int opcode;

      init_table();

      while (true) {   /* loop forever */

      opcode = next_instr();   /* Assume next_instr will
                                  return    the     next
                                  intermediate code */

      (*jumptable[opcode])(); /* branch to corresponding
                                 routine   according  to
                                 opcode */

      }
}
```

Figure 3.5 Jump Table In C On Xenix

For the sake of simplicity, three large Xenix sequential files were used to emulate the two floppy disk drives and printer of the Edison system. Therefore,

the I/O drivers part of the kernel became very easy to be implemented, thus this kernel program was simpler than the original one. Several utility programs were developed along with this design in order to access the above mentioned virtual floppy disks. After the kernel program was completed, more typing and programming errors of the source codes were found. Although it took a couple of more weeks to accomplish the above extra implementation, this high level kernel program was found to be very helpful in improving the correctness of the system software.

3.2.3 Edison Kernel In 8086

After the above phase had been completed, the kernel program was then rewritten in order to port the Edison system to the 86/310 microcomputer. This kernel program was implemented in 8086 assembly language by using the ASM86[Irmx83] assembler, and the working environment was then changed to the IRMX/86 system which runs on the 86/310. The IRMX/86 operating system is designed for use with programs executing on the iAPX86, iAPX88 and iAPX286 microprocessors[Irmx84]. At that point, it was necessary to implement not only the threaded code interpreter but also the I/O device drivers and interrupt handlers. In order to gain direct control of those two floppy disk drives, it was

necessary to program the Intel 218A floppy disk controller card[Isbx83], and this was the most time consuming programming work that had been encountered in this porting experiment .In addition, these two floppy drives in the 86/310 machine are not identical, their maximum capacities are 360K and 720K bytes respectively. Since a symmetrical floppy disk system may provide a more user friendly working environment, extra codes were added to the driver routine in order to adjust this difference and then virtually these two floppy disk drives could be operated like two identical 360K drives. This trick enabled both disk drives to access 360K-byte formatted floppy disks. Another important issue that must be considered was the size of the addressing unit. In the 86/310 machine, the word addressing memory store configuration was not suitable for the kernel program. The following 8086 assembly routine explains the reason:

```
NEXT_INSTR :
   MOV BX,  [DI]            ;  Register DI serves as the program
                            ;  counter. Register BX contains the
                            ;  intermediate  code  needed to be
                            ;  executed after  this instruction.
   SHL BX,   1              ;  Shift left to obtain the offset
                            ;  of corresponding routine.
   JMP WORD PTR OPBASE[BX]  ;  Jump to the routine.
```

Figure 3.6 Sample 8086 Assembly Routine

The routine in Figure 3.6 will be executed whenever the kernel needs to fetch the next intermediate code instruction. It is obvious that if the Edison compiler generates byte addressing intermediate codes, the SHL instruction can be eliminated and the overall performance of the kernel will be improved. For this reason pass 4 of the Edison compiler was rewritten and all the Edison programs were recompiled to get the new version of their intermediate codes.

3.2.4 Edison System On 86/310

When the Edison system was ported on top of the Xenix system, the virtual Edison disks could be initialized by using the afore-mentioned utility programs. This time another method was used to do the system transportation since the disks were not simulated by data files. The first idea that came up was to use a IBM PC to directly access a 86/310 floppy disk since they both use the same type of common 8272 disk controller.

Method 1. Load data through the disk

The data transportation procedure was then as follows :

1. In 86/310 use Xenix **Doscp** utility program to load a Edison data file to a IBM PC DOS disk.

2. Insert the above disk into a IBM PC, use IBM **debugger** program to load the Edison data file to the ram.

3. Insert a blank disk into the IBM PC, use the **debugger** program again to load back the data from the ram to specific sectors of the blank disk.

The above method was expected to allow the implementator to create an Edison disk without much trouble. However, this method did not work very well. The IBM debugger always assumes that each track of the blank disk will contain only 8 instead of 9 sectors. It means that the debugger will always skip sector 9 when data is loaded to the disk from the ram. The solution towards this problem was using a disk sector editor to adjust the difference. The sector editor program was written in 8086 assembly language and run on the 86/310 machine. Two more steps were then added to the transportation procedure.

4. Take the blank disk to a 86/310 machine, and use the sector editor to fill up sector 9 of each track with correct data manually.

5. Use the sector editor to update the diskmap and diskcat of the Edison disk.

The intermediate code file of the Edison OS was loaded to the disk by the above method without much trouble. After the Edison OS was up and running on the 86/310, the rest of the Edison software was expected to be transferred to the 86/310 machine by using the same procedure. However, it was then found that the above method was not feasible for transferring large Edison source files. The adjustment that had to be taken for each file transfer was just too complicated. When the Edison OS code was loaded to the disk , there was no need to modify the diskcat since the OS code was fixed to reside in sector 6 to 21 of the disk. In addition, there was no need to store the OS code file entry in the diskcat. Based on experience , it might take hours to transfer just one single Edison source file by using the above method. In order to avoid the above mentioned time consuming operations, an alternative method was derived to carry out the data transfer.

Method 2. Load data from serial port

The second idea was to use a serial channel to perform the data transfer. The additional hardware that was required for this method was a serial communication card which could be connected to the iSBX bus[Isbc82a] of the 86/310. The following circuit diagram shows the

details of the serial communication card.

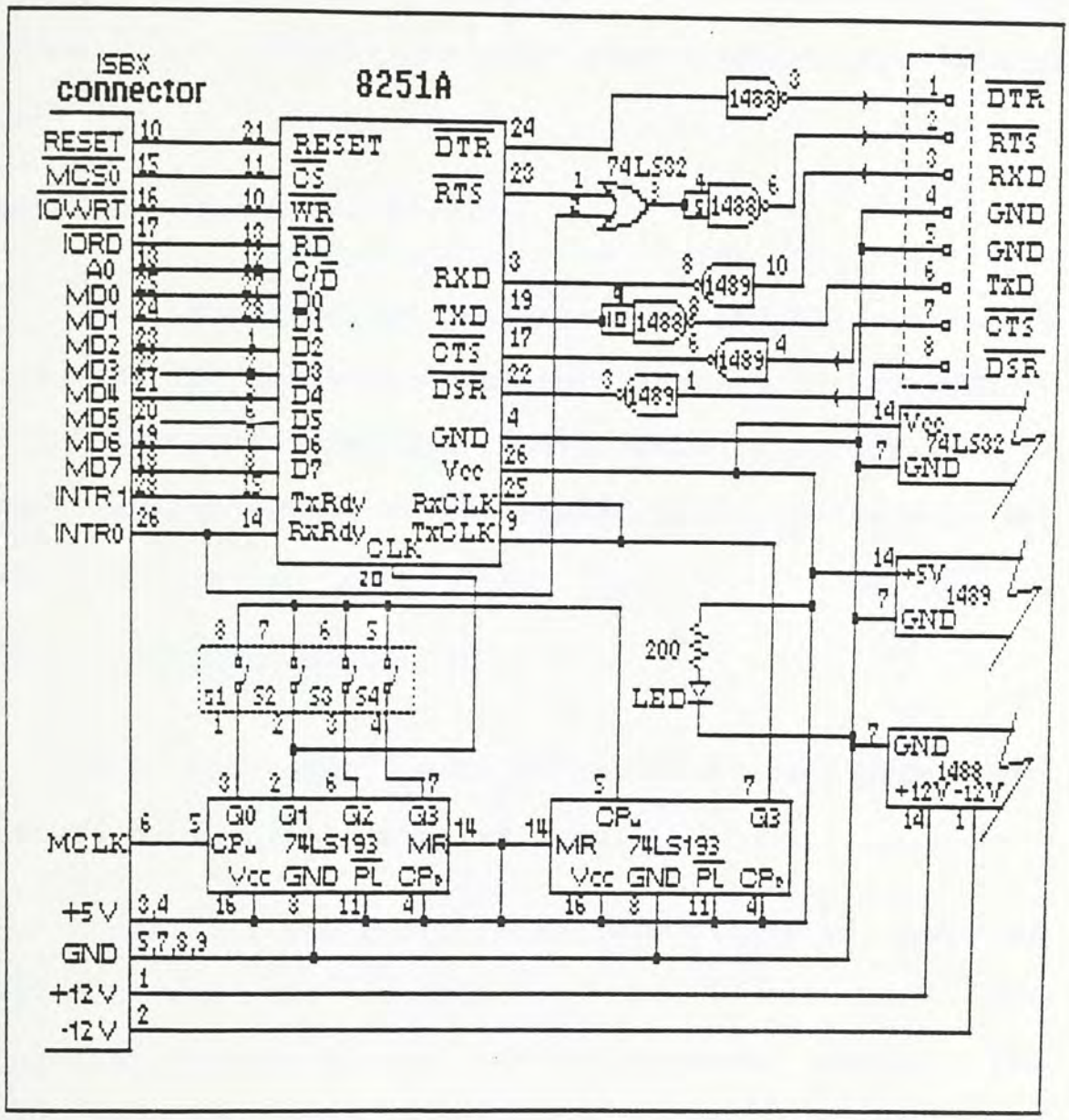


Figure 3.7 Serial Communication Card

A corresponding I/O routine was added to the kernel program and an access method was designed to allow application level programs to interface with this

new serial I/O routine. This method allows the application programs to access the I/O port directly through the standard Edison kernel call **sense** and **obtain**. The usages of these kernel calls are listed below :

sense(portaddr,bitpattern):

In this function call,portaddr and bitpattern are integers. The notation [portaddr] represents the content of I/O port portaddr. The return value is a boolean type value which depends on the result of ([portaddr] AND bitpattern).

obtain(portaddr,data):

The [portaddr] will be loaded to data after the execution of the above kernel call.

After all the modification was completed and the serial card was installed, the serial card was connected to one of the 286/310 terminal ports. The data transfer then became very easy, and the data received by the application program was stored in the file system by standard system file routines. Since the diskmap and diskcat was maintained by the operating system, there was no need to modify the diskcat or diskmap manually.

3.2.5 Transporting Edison To 286/310

After the porting to 86/310 was finished, a new kernel program was implemented in order to port the Edison system to the 286/310 microcomputer. Since the iAPX286 microprocessor is functionally identical to iAPX86 when it is operated in real mode, the new kernel program was also written in 8086 assembly. Therefore the threaded code interpreter part of the kernel program was not required to be modified in any way. However, all the I/O drivers and interrupt handler routines must be rewritten since the hardware configuration of 286/310 is totally different from the hardware configuration of 86/310. The 286/310 is equipped with one 14 Mega-byte hard disk and a 360K floppy disk drive. It was obvious that the original single level Edison file system was not able to cope with this hardware configuration. Since a new file system was not yet developed, only the first 360K memory of the hard disk could be addressed by the Edison system at that stage. Generally speaking, this part of the porting was easier than the previous one since all the system software had already been proved correct on the 86/310 machine. Therefore, less than two man-months were taken to complete this part of the porting. In addition, a serial communication card was also added to 286/310 machine through the iSBX bus. This

card was installed for the purpose of distributed processing.

3.3 MEASUREMENT OF PORTABILITY

One way to measure the portability of a particular system may be based on the time it takes to transport the whole system. The following list is a chronology of the 86/310 porting experiment.

The project was begun in Sept. 1985

1. It took about one man-month for understanding the Edison system. At the same period of time the source text of the Edison programs (Os, Compiler) were typed into the development system(Xenix 286).

2. It took about 2 man-months to finish the coding phase of the bootstrap compiler.

3. Another 6 man-weeks were spent to complete the Edison system which ran on top of Xenix. This included the debugging period. Often it was very difficult to locate an error, since errors may occur in the source, compiler or kernel itself. The system was up in April, 1986. Another one man-month was spent in writing up some more Edison programs for system testing.

4. The following 9 man-weeks were spent on the

implementation of the I/O drivers for the 8086 kernel program. Another 3 weeks of the time were spent on the construction of the threaded code interpreter.

5. After the kernel program was completed, the Edison files were then transferred to the new system . It took 5 man-weeks to make the new Edison system running.

6. For the purpose of final testing, three man-weeks were spent to construct a simple line oriented editor for the new Edison86 system. In Oct. 1986. the porting project was finally finished. All of the above work was based on part-time basis(18hrs/week).

3.4 MODIFICATION AND ENHANCEMENT

Before the intermediate code of the Edison Operating System could be transferred to the new machine, the file system of the Edison system must be modified first. The original Edison file system assumes that the capacity of the floppy disk is only 256K bytes , but the new machine has two 360K bytes floppy disk drives instead. Another major difference is the boot-up procedure. For simplicity, the booting job of Edison86 is performed by the IRMX86 bootstrap loader, meaning that the kernel code must reside in a IRMX86 boot disk as an executable file. After the machine is brought up

to a known state, the kernel would be executed and take control of the hardware. At this point the user has to insert a Edison format disk containing the OS code of the system which will then be loaded into the memory by the kernel. Therefore, there is no need to store the kernel code in the Edison disk. Notice that this modification does not decrease the efficiency of the Edison system and also save up more disk space. The following table lists the differences between the original version and 310 version of Edison system.

	Original	Edison86
Logical disk capacity	250k byte Page 1- 249	360k byte Page 1 - 360
Sectors per page	8	2
Bytes per sector	128	512
Starting Sector of disk map	10	0
Size of disk map	4 sectors	2 sectors
Starting Sector of disk catalog	14	2
Size of disk catalog	12 sectors	4 sectors
Maximum file entries per disk catalog	47	63
Starting sector of Kernel code	26	no kernel code
Starting sector of OS code	90	6
Serial Communication	not provided	provided

3.5 SUMMING UP THE PORTING LESSIONS

Edison is a portable operating system, i.e., one that can be transported with less effort then writing it from scratch. Based on the porting experience, it is estimated that a complete porting of Edison system may take one man 4 months(full time), starting with no knowledge of Edison. It would seem that the system is highly portable because it takes much less effort to transport than to rewrite[Brin83]. The Edison system is also flexible enough to fit in different environments. In this porting experiment, although the file system size is changed from 256K to 360K, the addressing unit is changed from word to byte; the lines changed in the source files for the entire project are only 1 or 2%. There are two important issues that must be considered. The first is the **overall portability** of a portable system, the other is the portability of the file system.

As mentioned at the beginning of this thesis, portable systems are attractive since they can provide well-tested system and application software economically. Therefore, a complete system porting must include the transfer of both the system and application software. When the overall portability of any system is being measured, two issues must be considered. The first

one is how difficult the original system software can be made to run on a new machine with minimal modification; the second one is how easy we can transfer all the application software of the original system to operate in the new environment. What Edison really needs is a **COMMUNICATION CHANNEL** to let application software to be transferred to other new Edison system. In this project the **COMMUNICATION CHANNEL** is a serial communication card. A more general solution may be adding standard Edison kernel calls to allow the operating system to interface with other systems, thus the **overall portability** of Edison can be increased. Therefore, it is suggested that a standard communication channel must be included in the basic design of any portable system.

When the Edison system was ported to the 286/310 microcomputer, the original design of the Edison file system was found unable to be adapted to a hard disk system. This means that the Edison file system is also machine dependent. In order to increase the system portability, a more flexible new file system must be designed.

3.6 SUMMARY

We have shown how a Edison system is ported to two different types of modern microcomputers. The Edison

kernel has to be modified to improve overall portability of the Edison system. No more than 2% of the os and compiler source need to be changed in any way. Generally speaking, the speed of the system in both 86/310 and 286/310 environment is quite acceptable. Although the disk I/O speed of the 86/310 system is not yet satisfactory, the system is operable. Besides, the speed problem of portable systems could be overcome as more powerful microprocessors are developed.

CHAPTER 4.

EDISON RING

This chapter describes the basic structure of the Edison Ring. The Edison Ring is a variant of the register insertion ring[Andr81] with a raw data rate of 9600 bits/second.

4.1 OVERVIEW

Edison machines are in this project interconnected by bit-serial communication channels to enable distributed processing. The bit-serial interconnect scheme is attractive since it allows the user to interconnect microcomputers made by different manufacturers and often with different internal architectures[Cay80]. The bit-serial communication links are implemented using RS-232C interfaces. Each Edison machine is designed to be connected to its two neighbours to form a double loop communication network. In order to handle transmission and reception of data streams between nodes with minimum errors, interrupt handlers(one for each channel) in each node are responsible for the capture of data transmitted from adjacent nodes.

4.1.1 Hardware Configuration

Serial communication between Edison nodes is accomplished using an external serial card based on two Intel 8251s. The hardware configuration is similar to the one mentioned in the last chapter. The only difference is that this serial card contains two sets of hardware components.each of which provides a full duplex communication capability. The serial cards are connected to the 86/310 and 286/310 through the iSBX bus. The data transmission between the cards is over twisted-pairs. The receive ready (RxRDY) pins of the Intel 8251s are connected to the interrupt pin of the PIC 8259A of the main board computer. For the 86/310, the RxRDYs are connected to interrupt pin 0 and pin 3; for the 286/310, the RxRDYs are connected to interrupt pin 1 and pin 2. The communication hardware is operating at the following modes:

Transmission rate: 9600 Baud
Byte length: 8 bits
Number of stop bits: 1
Parity: none

The figure in the following page describes the structure of the network.

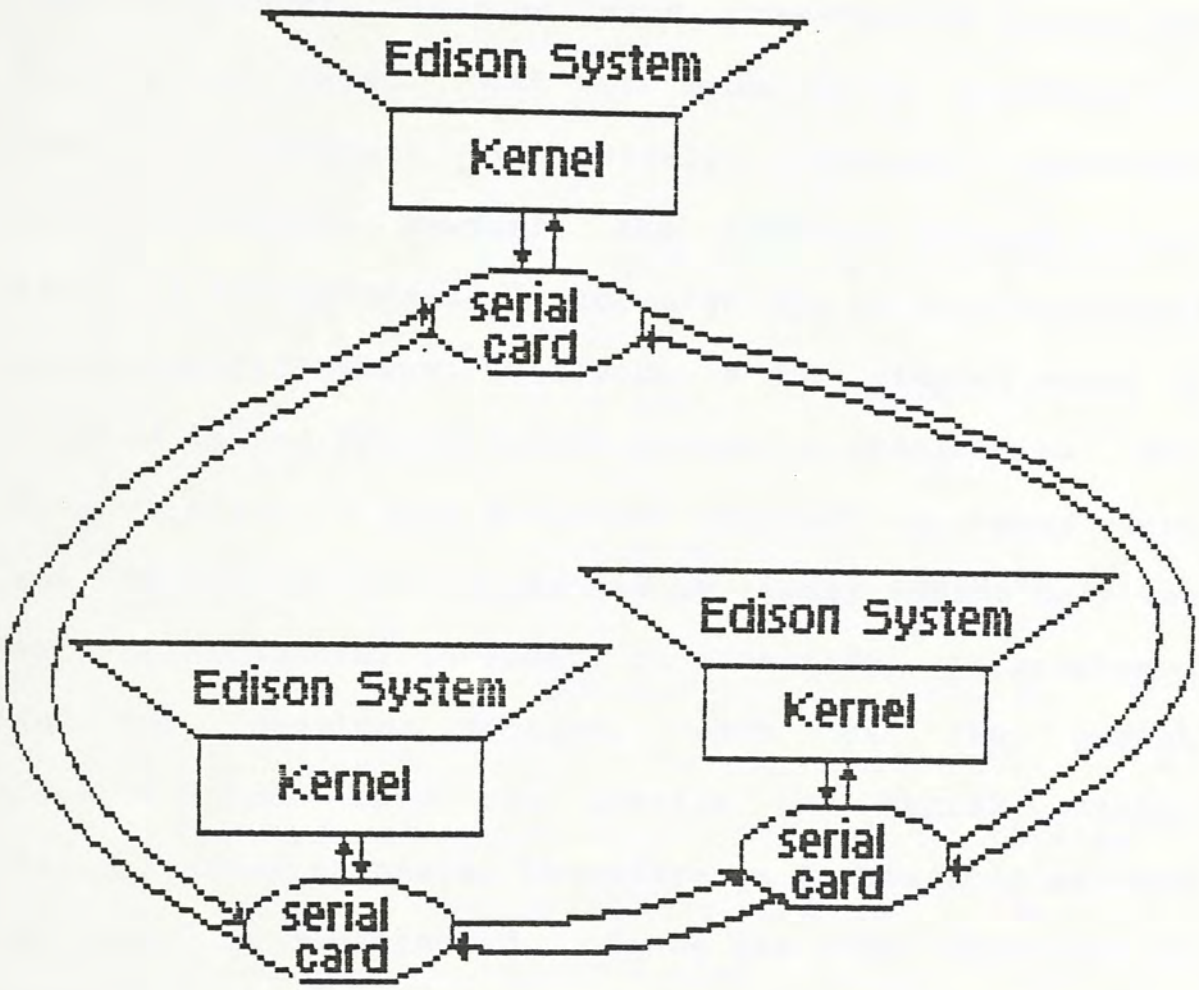


Figure 4.1 Network Structures

4.1.2 Design Issues

The Reference Model of Open Systems Interconnection (OSI) is a widely discussed framework for message-passing systems[Andr85]. The model has seven

layers, each performing a well-defined function to support the network service[Zimm80]. The seven layers are the physical layer, data-link layer, network layer, transport layer, session layer, presentation layer, and application layer. With this model it is possible to connect computers with widely different operating systems[Andr85]. However, the overhead created by all these layers generally is too high for an interconnected microcomputer system. Therefore, a much simpler model is adopted in the hybrid Edison system to provide low cost communication. A loop structure approach is chosen since the design can be implemented by simply adding only two sets of bit-serial interface to each node. As mentioned in the previous section, each of the serial communication card can provide two serial duplex communication channels, therefore, a double loop network structure is constructed. Since the major objective of the research is to identify the issuses of portability, several assumptions have been taken. The communication channel between nodes are assumed to be error free, ie. there will be no corrupted or lost frames. Actually, since the two hybrid systems are only a few meters apart and there is no external generated noise to affect the channels, no corrupted or lost frames have been found during the experiment.

4.2 RING STRUCTURE

A loop structure configuration is attractive in a multimicrocomputer system[Cay80]. Comparing with carrier sense networks , the major advantage of a loop is low initial cost in the configuration. The actual cost is always proportional to the number of machines that are to be interconnected. Moreover, loops provide for easy implementation of distributed switching mechanisms without the need for any sophisticated common control because each pair of interconnected loop interface can provide its own synchronization. Unlike some other ring networks, there is no independent ring IMPs(Interface Message Processors)[Andr81] on the Edison ring. The control needed is taken care by the microcomputer itself. Therefore, an easy implementation of distributed switching mechanism implies a simpler software routine is required to be implemented in the kernel program to support the control of the ring. If the interface mechanism is complicated, it will directly affect the portability of the system. Thus, ring structure communication is very attractive in a hybrid system.

4.2.1 Register Insert Ring

The register insertion ring is a more sophisticated version of the slotted ring[Andr81]. Since

the Edison ring is a variant of the register insertion ring. the basic design of such ring is discussed in this section.

In a register insertion ring, each ring interface contains two registers, a shift register and a output buffer. When a node has a packet to send out, the packet is first loaded into the output buffer. The packet size may be of variable length , it is only limited by the size of the output buffer.

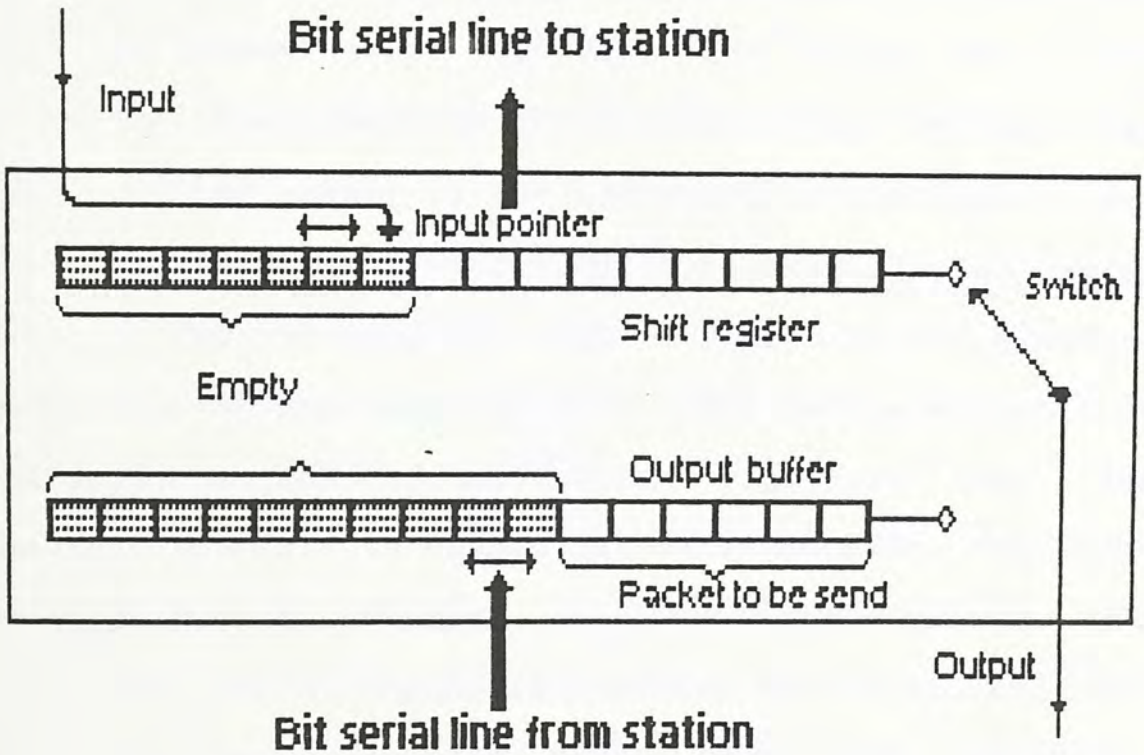


Figure 4.2

Interface Of A Register Insertion Ring[Andr81]

When the ring is started up, both of the interface registers are assumed to be empty. The corresponding input pointer shown in Figure 4.2 is then pointing to the rightmost bit position in the shift register. When an incoming bit is received from the ring, it is stored at the position pointed at by the input pointer, and the pointer is then moved 1 bit to the left. When the address field of the incoming packet is received , the interface can determine the destination of the incoming packet. If the destination address match the current node, the rest of the packet is redirected to the node, and the packet is removed from the ring. The input pointer is then reset to the rightmost bit. On the other hand, if the packet is not addressed to the local node, the interface simply forwards it to the adjacent node. Each of the incoming bit will be placed in the position pointed at by the input pointer. The entire contents of the shift register is then shifted right one bit, thus the rightmost bit is pushed out onto the ring. The input pointer is not advanced. If no new input arrives, the rightmost bit of the shift register can be shifted out and the input pointer moved right one position. Eventually, the entire packet will be transmitted out onto the ring by the shift right operation. Whenever the shift register has pushed out the last bit of a packet,

the interface will check the output buffer. If the output buffer is not empty, and the number of empty slots in the shift register is at least as large as the output packet, the output switch is flipped and the output buffer which contains the output packet is now shifted out onto the ring, 1 bit at a time, in synchronization with the input[Andr81]. The major advantage of the register insertion ring is that it allows multiple packets on the ring.

4.2.2 Edison Ring

The major difference between the Edison ring and the register insert ring is that the Edison ring packet transmission is done in an asynchronous manner. Since each of the Edison node is assumed to be a single processor machine , the ring interface is integrated within the kernel program. The overhead created by ring synchronization is substantial, therefore, asynchronous ring communication is more preferable. Both of the Edison ring and register insertion ring allow its packet to be variable length. These two rings also normally operate multiple packets, and there are multi-bit delays during the data transmission.

The ring interface contains 5 buffers as shown in Figure 4.3. Two boolean semaphores are associated with the two channels. When a node has a packet to transmit,

the packet is first loaded into the extra buffer. The packet size may be of variable length, up to the size of the extra buffer. A netserver module is defined and integrated within the kernel program, the function of the routines in this module is to monitor the data communication the of Edison ring.

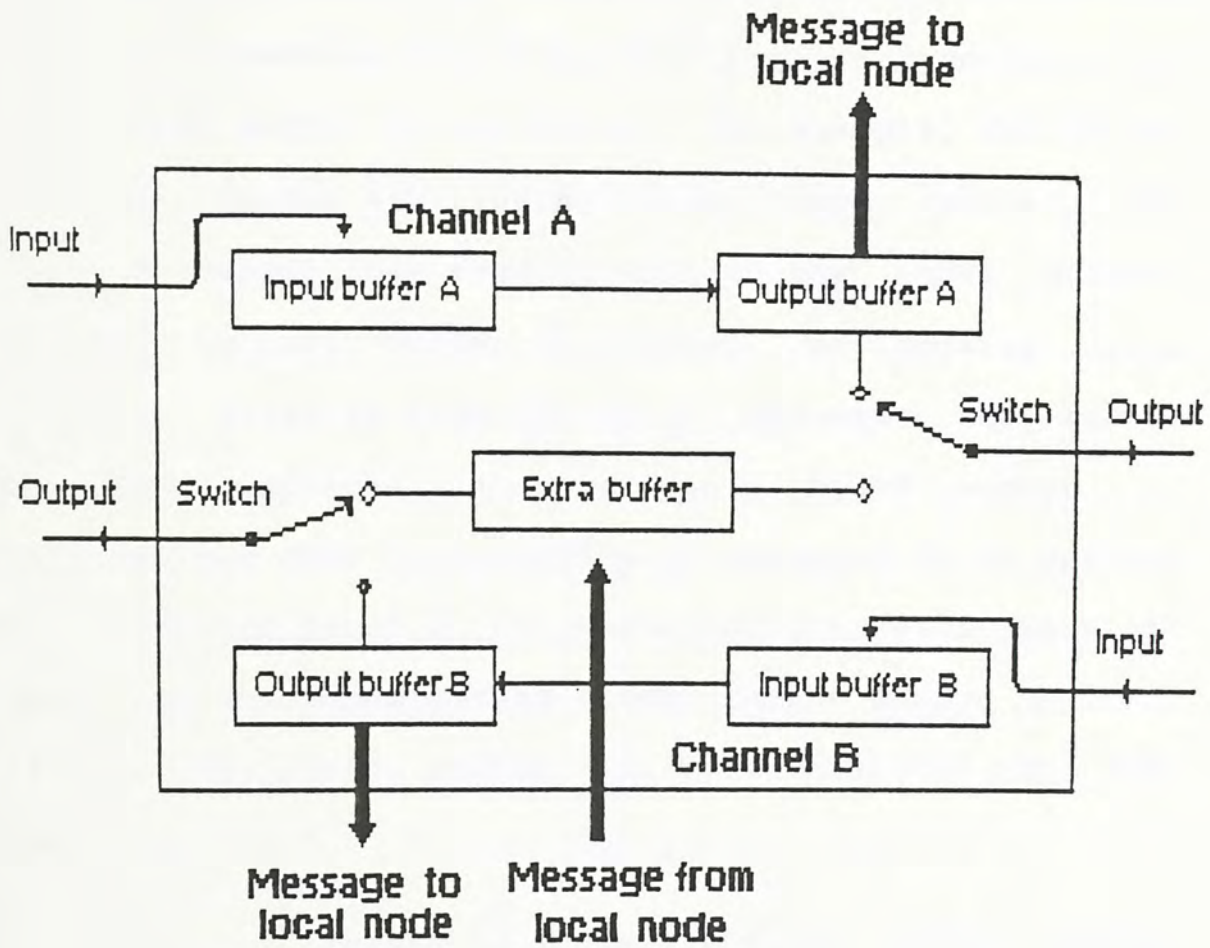


Figure 4.3 Interface Of Edison Ring

When the ring is started up, all the buffers are assumed to be empty and both of the boolean semaphores are set to true. All buffers will store data in a first in first out manner, the corresponding index pointers are used to ensure this policy. When an incoming byte is received from the ring, it is placed in the corresponding input buffer. After the rest of the packet has been received, the whole packet will be loaded into the corresponding output buffer if it is possible. If the output buffer is not empty, for example, due to the previous packet still being in the output buffer, the current packet may need to wait in the input buffer. Before the input buffer is cleared, no incoming bytes will be accepted from the ring. Whenever the kernel program completes interpreting a fixed number of intermediate code instructions or whenever it is waiting for keyboard input , the kernel program will check if there is an output packet placed in the output buffer. If so, the output packet will be transmitted out onto the ring.

When a packet is loaded into the output buffer from the input buffer, the interface determines whether or not this packet is addressed to it by examining the address field. If so, a netserver routine will be called up to serve this packet. This packet may either be a

reply packet or a request packet. If it is a reply packet, the netserver routine extracts the response, inverted the corresponding channel semaphore, and removes the packet from the ring. Otherwise the netserver routine will serve the request packet. During the service , the contents in the packet may be updated by the netserver routine. For instance, the destination address may be modified to its source address in order to send the packet back to it source node. When the service is completed, the packet which now is containing the response will be transferred onto the ring.

When a node wants to send out a packet, the packet is first placed in the extra buffer. The interface routine checks the two boolean semaphores to determine which channel can be used to transmit the output packet. In other words, if the value of channel A semaphore is true, the packet in the extra buffer can be transmitted out onto the ring through channel A, and the corresponding semaphore is inverted. Until one of the channel is available, the output packet must wait in the extra buffer. It is now clear that the semaphores are used to avoid communication dead lock[Andr85]. If there are N machines interconnected to from an Edison ring,the maximum number of packets in each channel will equal to N. Since the total number of buffers(input/output) is 2N

in each channel, the circular wait condition never holds, thus the dead lock condition can be avoided[Jame83].

4.3 THE PACKET PROTOCOL

All communication round the Edison ring takes the form of packets consisting of up to 538 bytes of data. The first 13 words of the packet contain the header and the rest of the packet contains data. Thus the rest of the packet can carry 512 bytes of data, which equals to the size of a disk sector. The format of the packet is defined as follows :

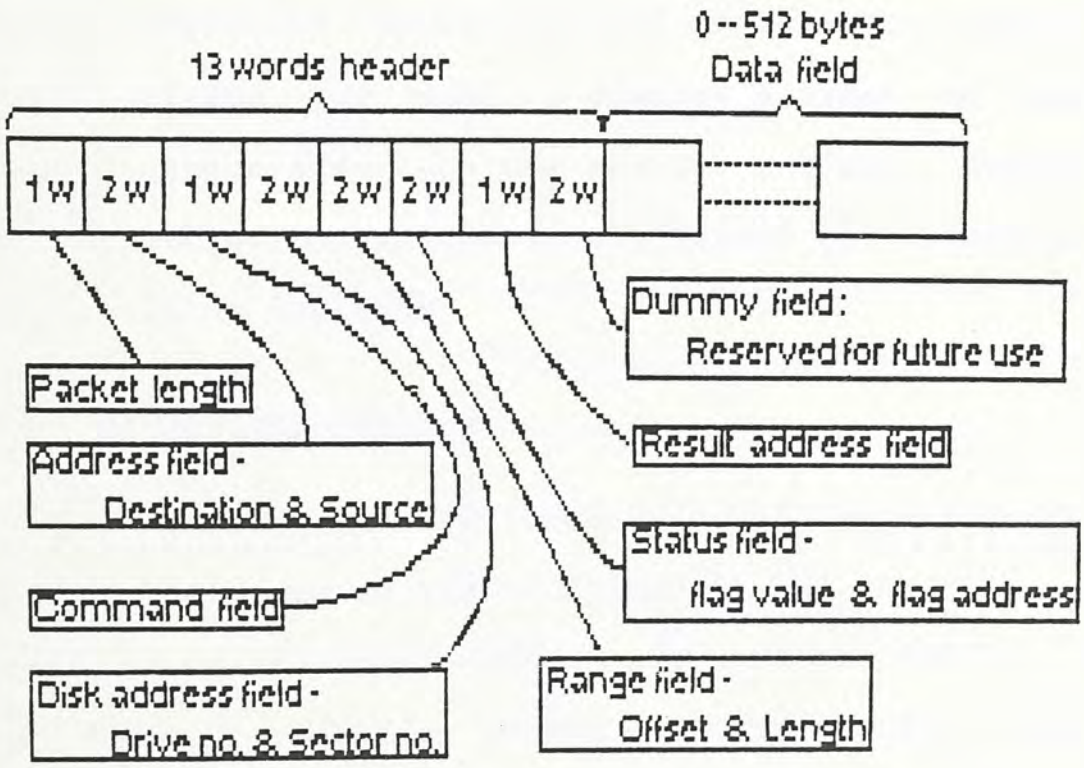


Figure 4.4 Packet format

4.3.1 Packet Format

There are three types of packet, namely the request packet, the reply packet , and the broadcast packet. All three types of packet use the same packet format as described in figure 4.4. Since packets may have variable length, the first word of the packet contains the length of the entire packet. The ring interface may use this value to determine the boundary of packets. The command field is used to specify the request. The drive and sector fields are used to contain associated drive number and disk sector number if the remote request is related to disk access. The offset and length fields are used to specify a range of memory which is associated with the remote request. Functions of the rest of the fields in the header are described in the following sections.

4.3.2 Naming And Routing

A unique 16 bits integer number is assigned to each of the nodes as its node identity in the ring. The range of this integer number is from 1 to 65535. Thus the maximum possible number of nodes that can be interconnected is 65535. The value zero is reserved in the case that a packet must be broadcasted to all nodes by setting the destination address to this value. The

routing scheme is simple. The message can choose either channel A or B to follow in reaching its destination, and the reply packet will return to its source through the same channel. When a request packet is received by the destination node, the netserver of the destination will complete the request and put the result in the data field of the packet. The destination field of the packet is modified to equal to the source , and a request complete flag will be placed in the result flag field. In case that the request is not able to be satisfied, the netserver will place an appropriate value in the result flag field to indicate the result of the operation.

4.3.3 User Interface

As described in the previous sections, the ring interface is actually composed of a netserver module and a interrupt handler. A kernel call, 'remote' , is provided to allow the operating system to interface with the ring. The operating system is responsible to hide the hardware details of the ring from the user/application-level program. For example, if a node wants to access a block of data, the node will make an ordinary system call no matter whether or not the required data is located in the local node or not. The operating system then has to determine where the data is

located and send an appropriate request to other node if necessary. Thus the transparency of distribution is ensured.

4.3.4 Separate Request And Response

When a node issues a request packet on to the ring through a system call, it may be possible for the caller to perform some other functions while waiting for the callee to return. The Edison ring mechanism separates the request and response, using a result flag in the caller. To initiate request , the source needs to specify a result address and a result flag pointer in the request packet. Therefore, when the reply packet is returned , the netserver can place the response in the memory location according to the result address. At the same time the result flag is placed in the memory location according to the result flag pointer. By examining the appropriate result flag value, the caller can determine whether the particular request is completed or not. With the afore-mentioned mechanism, the operating system can load remote file pages into the local file buffer in advance.

4.4 Summary

The Edison ring is a low cost asynchronous communication mechanism that can be used in a distributed heterogeneous system. The major advantage of this ring structure design is allowing incremental growth of the system with the minimum hardware investment. Although the data transmission rate in the hybrid system is relatively slow, however, since the required hardware and software is relatively simple, it would seem that such design can be easily implemented on different types of microcomputers , thus the portability of this design is extremely high. Furthermore , as it is a point to point system, there is no need to use the same medium throughout the entire system, and this provides a more flexible system configuration.

CHAPTER 5.

DISTRIBUTED EDISON SYSTEM

In order to provide a transparent distributed operating environment to the user, the original Edison operating system program must be rewritten to cope with the Edison ring. The design of the new Edison operating system is presented in this chapter.

5.1 DESIGN ISSUES

The major objective of the research is to develop a distributed system which can provide run time executable code sharing service. In order to do so, a local Edison machine of the distributed system must be able to access a remote file which contains the intermediate code source. Obviously, with the original Edison file system the operating system will not be able to provide this service.

5.1.1 The Original File System

The original file system assumes that the Edison system will consist only of floppy disk drives. In order to provide a more efficient operation, the disk allocation table - 'diskmap' and the file descriptor table - 'diskcat' are kept in the main store and directly accessed by the operating system[Brin83]. The

problem with this design is that the size of the 'diskmap' and 'diskcat' must then be fixed, which apparently lacks any flexibility at all. Considering the fact that nowadays microcomputers may be equipped with a 150 Mega-byte hard disk and this will require 1.2 Mega bytes of main store to hold the 'diskmap' and 'diskcat' according to the original file system design. In such cases, it will not be feasible to port the Edison system to the new machine. In other words, the portability goal is totally defeated by the file system.

5.2 THE NEW SYSTEM

A new file system must be designed to overcome the problem mentioned above. The new file system must be flexible enough to support both floppy disk and hard disk drives. The new file system may also need to at least provide multiple access service of data files. That is, under the file system, a data file is allowed to be read by several programs simultaneously.

5.2.1 The Approach

In distributed systems, it is preferred to have a single global file system visible from all machines[Andr85]. The convenience of having a single global name space is obvious. When this approach is used, there can be only one 'util' directory for utility

programmes, one 'tmp' directory for all temporary files, and so on. When a program wants to use the Edison compiler it does something like the followings :

```
open('/util/compiler',EXE)
```

without knowing where the file is located. It is up to the operating system to determine the location of the file and arrange for transport of file contents as they are needed.

Basically, two sections of the original system should be modified. One is the operating system program and the other is the netserver module of the kernel program. The modules in the operating system program that define the original file system has to be taken out and substituted by a new set of modules. Service routines must be added to the netserver module in order to provide more remote service which are related to the new file system.

5.2.2 The File System

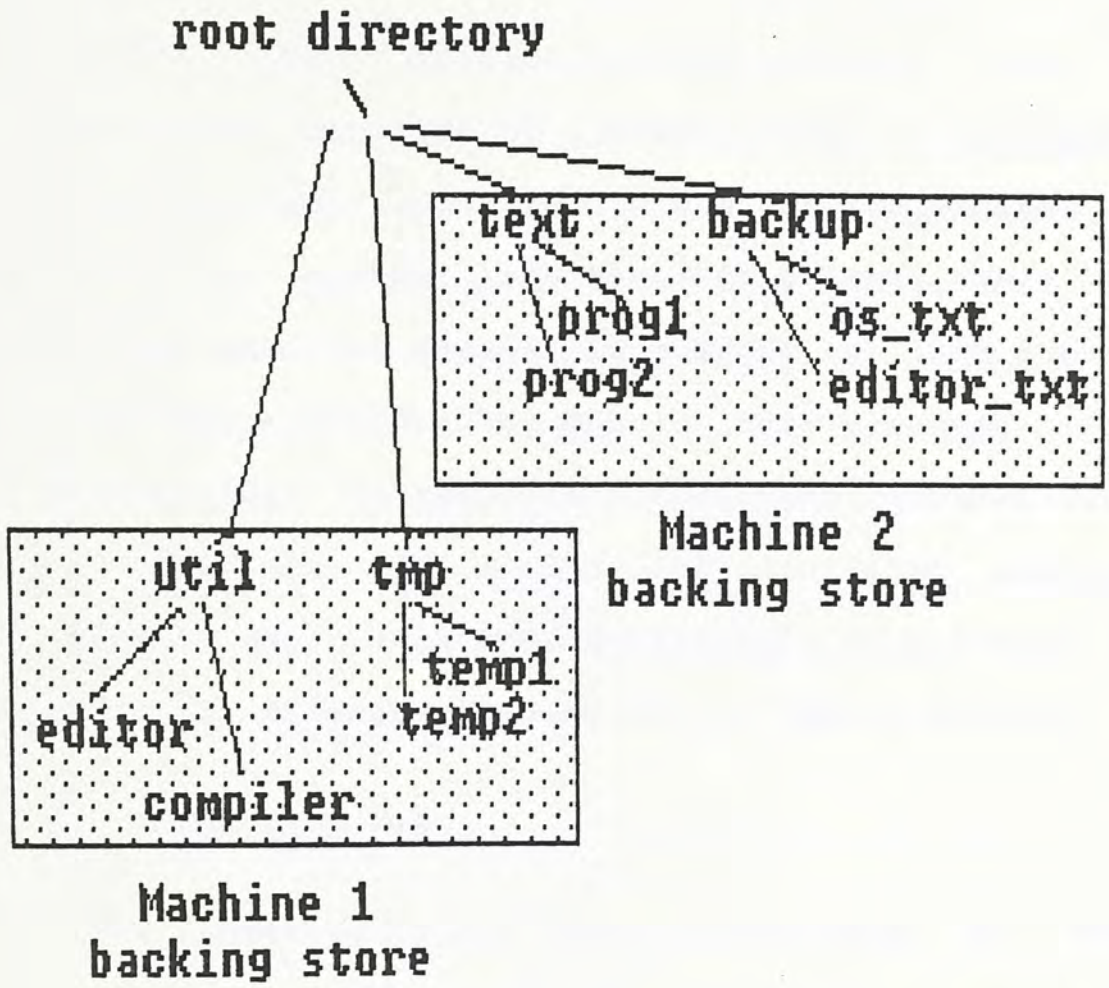


Figure 5.1 File Directory Structure

As shown in Figure 5.1 the new file system uses the multilevel directory approach. Files in the file system are accessed through a directory structure that takes the form of a rooted tree. The subtrees that are directly mounted on the root represent the backing store of each individual machine. Thus , remote directories and files can be accessed by simply referring to a

complete path name.

5.2.2.1 Directory table

Since the operating system program has to determine the location of a remote file, a directory table can be kept in the main store and maintained by the operating system. The contents of this table are initialized when the system is started up. Each machine may send out a broadcasted "Who's there" request on to the Edison Ring. In response, the other machines will return their node id and names of associated subtrees if there is any. This information will be gathered and stored in all directory tables of the Edison machines.

5.2.2.2 Allocation map

The backing store devices(subtrees) of each machine are treated as an array of 4 K-byte page of storage. The storage media may be floppy disk, hard disk, or tape. For each device there is a allocation map, stored at a fixed address, containing a description of the state and contents of each page on the device. The primitive operations allocate page and free page can be provided by service routines of the netserver module.

5.2.2.3 File descriptor

Files in the new Edison system are uniquely identified by their path name. Each file is associated with a file descriptor. The file descriptor contains the file name, status, and 10 page addresses which point to the first 10 data page in the file. If the file is larger than 40 K, all the rest of the data page will be chained up. This allocation algorithm is a simplified design of the Unix file system[Zarr81]. It is very simple, but it does slow access to large files. Directory entries are also different types of file descriptors.

The status of a file consists of several fields :

1. File type - An Edison file can be directory file, data file or executable file.

2. File size - The size is represented by a page number and an offset number.

3. Locks - A data file can be read by several programmes simultaneously, a read lock field is used to keep track of the number of readers. On the other hand, only one program is allowed to update a data file at any instance of time. A write lock field which contains a boolean value is used to ensure this policy.

4. States - A file may be in one of three states: normal, delete, pending. When a file is requested to be deleted , and the number of readers is not zero, the file will be marked to be in the pending state. Until all the readers have terminated(close the file), the file will not be deleted.

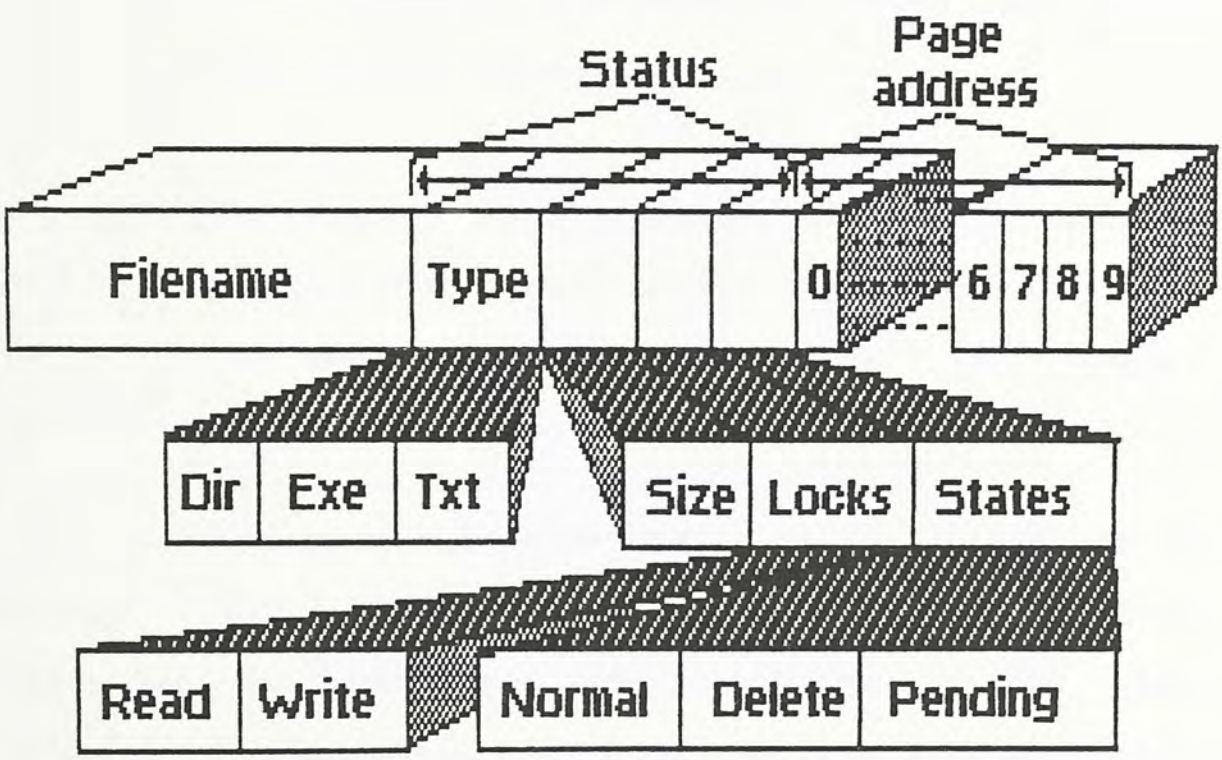


Figure 5.2 New Edison File Descriptor

5.2.3 Service Routines

In order to maintain the file system, the following Edison Ring service routines must be implemented in the netserver module of the kernel program.

1) Free page - This routine will modify the allocation map to release storage space. For example, when a data file is being deleted, all pages which are associated with the file must be released.

2) Allocate page - This routine will look up the allocation map and return a free page number to the caller and mark the page status to 'not empty' immediately.

3) Read/Write page - Provide the primitive operations to access a local or remote backing store. The parameters for these routines are a node id and the logical page number. By using these services, the read file and write file system calls can be easily implemented in the operating system program.

4) Create file - This routine is used to add a file descriptor to a directory file , the directory file may require to be expanded if necessary.

5) Open file - This routine returns the file descriptor to the caller. Since this operation involves directory tree search, it is better done in the kernel program. In addition, this routine is responsible for modifying the locks field of the corresponding file descriptor.

6) Close file - This routine may update the file descriptor when the file access is terminated. Normally, it will only decrement the locks count of the file. When the file is in the pending state and the locks count goes to zero, this routine will call free page to release the pages occupied by the file and modify the state field of the file descriptor to 'deleted'.

7) Delete file - This routine modifies the state field of the corresponding file descriptor to 'pending' and if the locks count equals zero, the file will be deleted immediately.

5.2.4 The New System Commands

Additional system commands are required to be implemented in order to let the users interface with the new operating system. Since the file system is changed to a directory oriented structure, the first additional system command may be 'cd' (change directory). The 'mkdir' (create directory) and 'rmdir' (delete directory) commands are also needed to be constructed.

Based on the service routines that have been mentioned in the previous section, these additional system commands could be implemented easily.

5.2.5 Edison vs Unix

Although the Edison file system design is based on the Unix file system, and both of them are portable systems, there are significant differences between these two systems in distributed environments. When connecting two or more Unix systems together , the user may use its network "USENET" to access remote files. This remote file transfer service is not supported by system calls, instead, the user must run a special file transfer program "uucp" to carry out the file transfer operation[Andr85]. On the other hand, the same service in the distributed Edison system will be supported by system calls and the operating system will hide all the details of the distributed system. Another distinctive difference is that the distributed Edison system provides direct executable code sharing while the Unix system does not.

5.3 THE OVERALL PORTABILITY

When compared with the original Edison system, it can be seen that the overall portability of the hybrid system is superior due to the following reasons:

I. The Edison ring allows system software to be transferred to a new machine directly. Once the kernel program is implemented in a new machine, the new machine can access all the system software immediately. Therefore, there is no need to spend extra time on solving the data transfer problem.

II. The new file system design can be applied to modern microcomputers equipped with large variable size backing store devices. On the other hand, microcomputers equipped with no backing store devices now can also be joined to the distributed Edison system family.

III. Since the new file system hides all the hardware detail of the secondary store devices, the operating system program is not required to be modified at all, thus no recompilation is needed.

CHAPTER 6.

CONCLUSIONS

The value of this research, to a large degree, depends upon the premises presented in Chapter 1: that software portability is of economic and reliable value, and furthermore that portable distributed system is of great flexible value.

The hybrid distributed Edison system demostrates the feasibility of using machine-dependent kernel program to support this portable distributed system. The kernel program of the hybrid system now consists of three independent modules, namely the threaded code interpreter module, the basic I/O module, and the netserver module. Each of these modules serves independent functions. It is believed that these modules are simple enough to be easily implemented on different types of microcomputers available nowadays. The general problem of portability and distributed heterogeneous computing environments is demonstrated and some of the answers have been supplied.

6.1 Further Enhancement

Several pragmatic enhancements would be suggested in the followings.

Remote procedure call is not included in this hybrid system because the focus of this research is on direct executable code sharing. Nevertheless, further enchancement of the hybrid system may be along the RPC issue. Since the coherence of the system is supported by the kernel program, remote procedure call may be easier to be implemented than other distributed systems.

The addressing space of the hybrid system is now limited to 64 K bytes [2.2.6 Memory Management]. This relatively small addressing space may somewhat guarantee system portability. However, complicated application software would seem to be impossible to built on this system and microcomputers available nowadays are usually equipped with more than 64K bytes of main memory. Therefore, further research on the issue of the addressing space is essential. Two ideas are suggested at this point. One way to enlarge the addressing space of the system is to modify the Edison compiler to generate 32 bits operand. The other way to overcome the limitation is the system kernel may load the library procedures into private 64K memory segments.

The design of the Edison Ring does not aim on making the hybrid system fault tolerant. The design issues involved the trade-off between performance and fault toerance and tended to get resolved in favor of

performance. Futher research in this particular topic will be an interesting and fruitful study.

As long as the economic and flexible promises of portable distributed system continue to exist, the research for better hybrid system and better techniques to implement those systems must go on.

REFERENCES

[Alan87] Alan Borning "Computer System Reliability and Nuclear War", Communications of the ACM, Vol. 30, No 2, Feb. 1987

[Andr81] Andrew S. Tanenbaum "Computer Networks", Prentice-Hall,Inc., Englewood Cliffs, N.J. 1981

[Andr85] Andrew S. Tanenbaum & Robbert Van Renesse "Distributed Operating Systems", Computing Surveys, Vol.17, No.14, p.419-470, Dec. 1985

[Brin83] Brinch Hansen, Per "Programming a Personal Computer", Prentice-Hall,Inc., Englewood Cliffs, N.J. 1983

[Caro80] Carollanne Ogdin "Tutorial:Microcomputer System Design and Techniques", The Institute of Electriacl and Electronics Engineers, Inc. 1980

[Cay80] Cay Weitzman "Distributed Micro/ Minicomputer Systems", Prentice-Hall,Inc., Englewood Cliffs, N.J. 1980

[Dai86] Dai, K.S. "Edison-80,A language for modular programming of parallel processes", Information Processing Letters 22,61-72,1986

[Davi80] David Taylor & Lyndon Morgan "High-Level Languages for Microprocessor Projects", The National Computing Centre, 1980

[Davi82] David R. Cheriton "The Thoth System: Multi-Process Structing and Portability", Elsevier Science Publishing Co., Inc. 1982

[Hami79] Hamish Donaldson "Designing a distributed processing system", Associated Business Press, London, 1979

[Irmx83] "ASM86 Language Reference Manual", order no.: 121703-003, Intel Corporation, Santa Clara, Ca., 1983

[Irmx84] "iRMX 86 Introduction and Operator's Reference Manual", order no.: 146194-001, Intel Corporation, Santa Clara, Ca., 1984

[Isbc80] "iSBC 215 Winchester Disk Controller Hardware Reference Manual", order no.: 121593-00Ø2, Intel Corporation, Santa Clara, Ca., 1980

[Isbc82] "ISBC 86/14 and ISBC 86/30 Hardware Reference Manual", order no. :144044-00Ø2, Intel Corporation, Santa Clara, Ca., 1982

[Isbc85] "iSBC 286/10A Single Board Computer Hardware Reference Manual", order no.: 147532-00Ø1, Intel Corporation, Santa Clara, Ca., 1985

[Isbx83] "iSBX 218A Flexible Diskette Controller Board Hardware Reference Manual", order no.: 145911-00Ø1,Intel Corporation, Santa Clara, Ca., 1983

[Jame83] James L. Peterson & Abraham Silberschatz "Operating System Concepts", Addison-Wesley Publishing Company, Inc., 1983

[John81] John K. Ousterhout "Medusa A Distributed Operating System" UMI Research Press, 1981

[John86] John S. Quarterman & Josiah C. Hoskins "Notable Computer Networks", Communications of the ACM, Vol. 29, No 10, Oct. 1986

[Micr85] "Microsystem Components Handbook V.I,II", Intel Corporation, Santa Clara, Ca., 1985

[Need82] R.M. Needham & A.J. Herbert "The Cambridge Distributed Computing System", Addison-Wesley Publishers Limited, 1982

[Norw79] Norwin Craef & others "How to Design and Implement Small Time-sharing Systems Using Concurrent Pascal", Software-Practice and Experience, VOL. 9, 17-24 (1979)

[Paul83] Paul J. J & Thomas S. H. "Transporting A Portable Operating System: Unix To An IBM Minicomputer", Communications of the ACM, Vol. 26, No 12, Dec. 1983

[Pete84] Peter J.L. Wallis "Portable Programming", Macmillan Publishers Ltd., 1984

[Rect84] Rector R. & Alexy G. "The 8086 Book", Osborne/McGraw-Hill general books

[Ritc78] Kernighan B.W. & Ritchie D.M. "The C programming language", Prentice-Hall, Englewood Cliffs, N.J. 1978

[Xeni84a] "Xenix 286 C Library Guide", order no.: 174542-002, Intel Corporation, Santa Clara, Ca., 1984

[Xeni84b] "Xenix 286 Programmer's Guide", order no.: 174391-002, Intel Corporation, Santa Clara, Ca., 1984

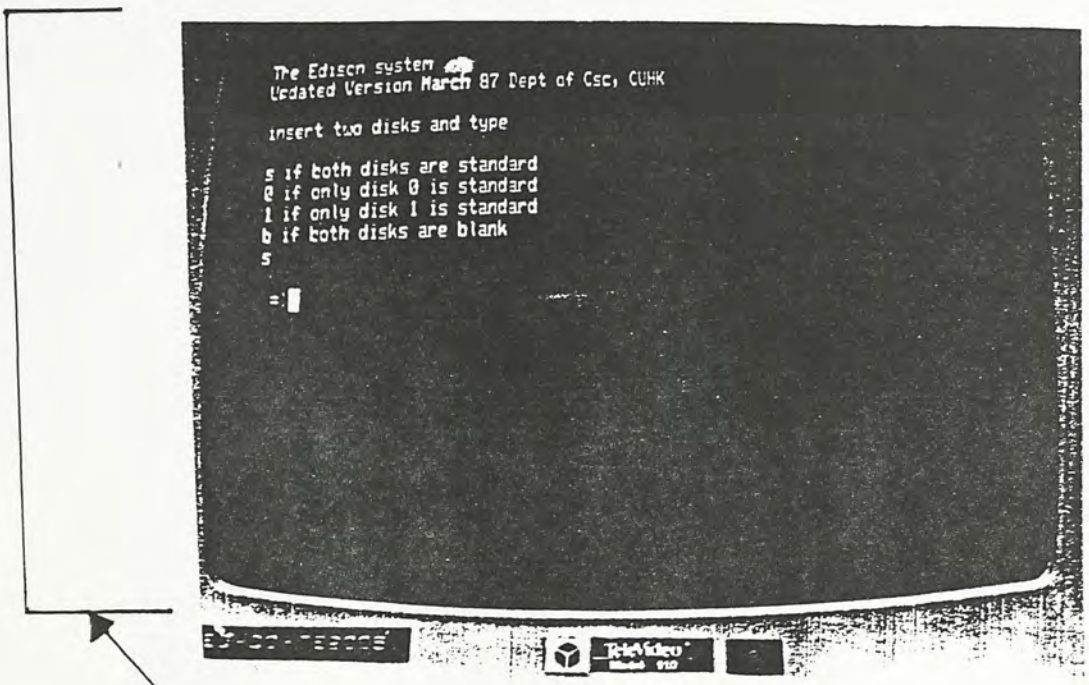
[Xeni84c] "Xenix 286 Reference Manual", order no.: 176279-001, Intel Corporation, Santa Clara, Ca., 1984

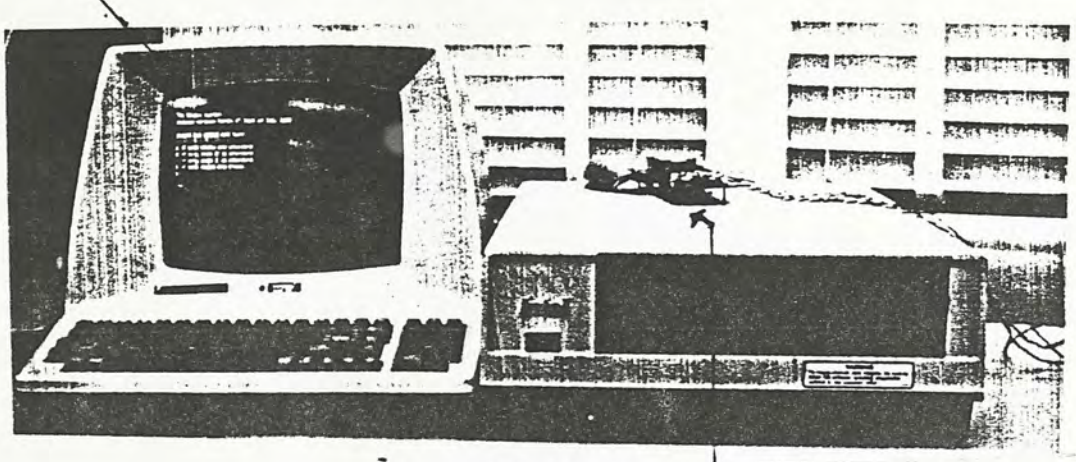
[Zarr81] Zarrella I. J. "Microprocessor Operating Systems", Microcomputer Applications, Suisun City, Ca., 1981

[Zimm80] Zimmermann, H. "OSI reference model-The ISO model of architecture for open systems interconnection", IEEE Trans. Commun. COM-28(Apr.), 425-432, 1980

APPENDIX A.

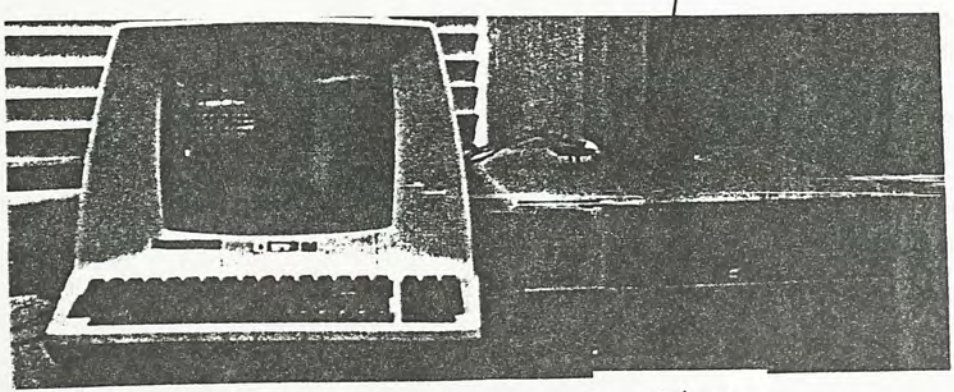
THE HYBRID SYSTEM





286/310

Serial communication cards



86/310

APPENDIX B.

PROGRAM SOURCE DISKS

1. The Edison compiler in C on Xenix :

2. The Edison kernel program in C on Xenix :

3. The Edison kernel programs in 8086

4. The Edison system disk 1 :

5. The Edison system disk 2 :